EP3260: Machine Learning Over Networks
Computer Assignment 1
Due Date: February 14, 2023

CA1 - Closed-form solution vs iterative approaches

February 14, 2023

Group 2
Siva Satya Sri Ganesh Seeram,
Hansi Abeynanda,
Eren Berk Kama,
Irshad Ahmed Meer,
Zinat Behdad

Let us consider

$$\mathbf{w}^\star = \underset{\mathbf{w}\in\mathbb{R}^d}{\text{minimize}}\ \frac{1}{N}\sum_{i\in[N]} \|\mathbf{w}^T\mathbf{x}_i - \mathbf{y}_i\|^2 + \lambda\|\mathbf{w}\|_2^2,$$

for a dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}$.

Then, address the following:

(a) Find a closed-form solution for this problem;

(b) Consider "Individual household electric power consumption" dataset ($N = 2075259$, $d = 9$) and find the optimal linear regressor from the closed-form expression;

(c) Repeat 2) for "Greenhouse gas observing network" dataset ($N = 2921$, $d = 5232$) and observe the scalability issue of the closed-form expression;

(d) How would you address even bigger datasets?

(a) Given objective function

$$f(w) = \frac{1}{N} \cdot \sum_{i \in N} \| w^T x_i - y_i \|_2^2 + \lambda \|w\|_2^2, \quad -(1)$$

For a closed-form soln. - w -

$\boxed{\nabla_w f(w) = 0}$ is required.

First we'll re-arrange (1) in terms of matrices X, Y

$$f(w) = \frac{1}{N} \cdot [w^T x_1 - y_1, w^T x_2 - y_2 \ldots, w^T x_N - y_N] \cdot \begin{bmatrix} w^T x_1 - y_1, \\ w^T x_2 - y_2, \\ \vdots \\ w^T x_N - y_N \end{bmatrix} + \lambda \cdot [w_1, w_2 \ldots w_d] \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

$$= \frac{1}{N} \cdot (w^T \cdot X^T - Y^T) \cdot (w^T X^T - Y^T)^T + \lambda w^T w$$

where, $w \in R^{d \times 1}$, $X \in R^{N \times d}$, $Y \in R^{N \times 1}$

d = dimensions/features
N = No. of samples/data

$$= \frac{1}{N} \cdot (w^T X^T - Y^T) \cdot (Xw - Y) + \lambda w^T w$$

$$= \frac{1}{N} (w^T X^T X w - w^T X^T Y - Y^T X w + Y^T Y + (N\lambda) \cdot w^T w)$$

$$= \frac{1}{N} [w^T X^T X w + w^T (N\lambda \cdot I) w - 2 w^T X^T Y + Y^T Y]$$

$$\therefore f(w) = \frac{1}{N} [w^T (X^T X + (N\lambda) I) w - 2 w^T X^T Y + Y^T Y]$$

Apply gradient wrt w

$$\Rightarrow \nabla_w f(w) = \frac{1}{N} [2 (X^T X + N\lambda I) w - 2 X^T Y + 0] = 0$$

$$\Rightarrow (X^T X + N\lambda I) w = X^T Y$$

$$\Rightarrow \boxed{w^* = (X^T X + (N\lambda) \cdot I)^{-1} \cdot X^T Y}$$

This is the closed form solution.

CA1 - Closed-form solution vs iterative approaches

February 14, 2023

Group 2
Siva Satya Sri Ganesh Seeram,
Hansi Abeynanda,
Eren Berk Kama,
Irshad Ahmed Meer,
Zinat Behdad

Part (b):Individual household electric power consumption

[1]:
```
##imports from libraries
import pandas as pd
import numpy as np
import time
from sklearn import linear_model
```

[2]:
```
## Load data and preprocessing:

## reading dataset ''household_power_consumption.txt'' :
## merging two first columns (date and time) in one column
data= pd.read_csv('household_power_consumption.txt', sep=';',
                  parse_dates={'dt' : ['Date', 'Time']},␣
 ↪infer_datetime_format=True,
                  low_memory=False, na_values=['nan','?'], index_col='dt')
```

[3]:
```
## Describe the data:
## count:  The number of not-empty values.
data.describe()
```

[3]:
```
       Global_active_power  Global_reactive_power       Voltage  \
count         2.049280e+06           2.049280e+06  2.049280e+06
mean          1.091615e+00           1.237145e-01  2.408399e+02
std           1.057294e+00           1.127220e-01  3.239987e+00
min           7.600000e-02           0.000000e+00  2.232000e+02
25%           3.080000e-01           4.800000e-02  2.389900e+02
50%           6.020000e-01           1.000000e-01  2.410100e+02
75%           1.528000e+00           1.940000e-01  2.428900e+02
max           1.112200e+01           1.390000e+00  2.541500e+02
```

```
       Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3
count      2.049280e+06    2.049280e+06    2.049280e+06    2.049280e+06
mean       4.627759e+00    1.121923e+00    1.298520e+00    6.458447e+00
std        4.444396e+00    6.153031e+00    5.822026e+00    8.437154e+00
min        2.000000e-01    0.000000e+00    0.000000e+00    0.000000e+00
25%        1.400000e+00    0.000000e+00    0.000000e+00    0.000000e+00
50%        2.600000e+00    0.000000e+00    0.000000e+00    1.000000e+00
75%        6.400000e+00    0.000000e+00    1.000000e+00    1.700000e+01
max        4.840000e+01    8.800000e+01    8.000000e+01    3.100000e+01
```

[4]:
```
## Find the number of 'nan' in each column:
data.isnull().sum()
```

[4]:
```
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
dtype: int64
```

[5]:
```
## Find the columns that have 'nan':
##(This section is not necessary since we can directly go through each column in␣
 ↪the next section)
droping_list_all=[]
for j in range(0,7):
    if not data.iloc[:, j].notnull().all():
        droping_list_all.append(j)
droping_list_all
```

[5]:
```
[0, 1, 2, 3, 4, 5, 6]
```

[6]:
```
## Replace the 'nan' cases in each column with the mean value of that column
## (in order to not change the stochastic parameters):
for j in range(0,7):
        data.iloc[:,j]=data.iloc[:,j].fillna(data.iloc[:,j].mean())
```

[7]:
```
## Define the first 6 columns as X:
X=data.iloc[:,0:6]
```

[8]:
```
X
```

[8]:
```
                     Global_active_power  Global_reactive_power  Voltage  \
dt
2006-12-16 17:24:00                4.216                  0.418   234.84
2006-12-16 17:25:00                5.360                  0.436   233.63
```

```
2006-12-16 17:26:00                5.374                 0.498   233.29
2006-12-16 17:27:00                5.388                 0.502   233.74
2006-12-16 17:28:00                3.666                 0.528   235.68
...                                  ...                   ...      ...
2010-11-26 20:58:00                0.946                 0.000   240.43
2010-11-26 20:59:00                0.944                 0.000   240.00
2010-11-26 21:00:00                0.938                 0.000   239.82
2010-11-26 21:01:00                0.934                 0.000   239.70
2010-11-26 21:02:00                0.932                 0.000   239.55

                     Global_intensity  Sub_metering_1  Sub_metering_2
dt
2006-12-16 17:24:00              18.4             0.0             1.0
2006-12-16 17:25:00              23.0             0.0             1.0
2006-12-16 17:26:00              23.0             0.0             2.0
2006-12-16 17:27:00              23.0             0.0             1.0
2006-12-16 17:28:00              15.8             0.0             1.0
...                               ...             ...             ...
2010-11-26 20:58:00               4.0             0.0             0.0
2010-11-26 20:59:00               4.0             0.0             0.0
2010-11-26 21:00:00               3.8             0.0             0.0
2010-11-26 21:01:00               3.8             0.0             0.0
2010-11-26 21:02:00               3.8             0.0             0.0

[2075259 rows x 6 columns]
```

[9]:
```python
## Define the last column as y
y=data.iloc[:,6]
```

[10]:
```python
y
```

[10]:
```
dt
2006-12-16 17:24:00    17.0
2006-12-16 17:25:00    16.0
2006-12-16 17:26:00    17.0
2006-12-16 17:27:00    17.0
2006-12-16 17:28:00    17.0
                       ...
2010-11-26 20:58:00     0.0
2010-11-26 20:59:00     0.0
2010-11-26 21:00:00     0.0
2010-11-26 21:01:00     0.0
2010-11-26 21:02:00     0.0
Name: Sub_metering_3, Length: 2075259, dtype: float64
```

[11]:
```python
X.describe()
```

[11]:

```
       Global_active_power  Global_reactive_power       Voltage  \
count         2.075259e+06           2.075259e+06  2.075259e+06
mean          1.091615e+00           1.237145e-01  2.408399e+02
std           1.050655e+00           1.120142e-01  3.219643e+00
min           7.600000e-02           0.000000e+00  2.232000e+02
25%           3.100000e-01           4.800000e-02  2.390200e+02
50%           6.300000e-01           1.020000e-01  2.409600e+02
75%           1.520000e+00           1.920000e-01  2.428600e+02
max           1.112200e+01           1.390000e+00  2.541500e+02

       Global_intensity  Sub_metering_1  Sub_metering_2
count      2.075259e+06    2.075259e+06    2.075259e+06
mean       4.627759e+00    1.121923e+00    1.298520e+00
std        4.416490e+00    6.114397e+00    5.785470e+00
min        2.000000e-01    0.000000e+00    0.000000e+00
25%        1.400000e+00    0.000000e+00    0.000000e+00
50%        2.800000e+00    0.000000e+00    0.000000e+00
75%        6.400000e+00    0.000000e+00    1.000000e+00
max        4.840000e+01    8.800000e+01    8.000000e+01
```

[12]:

```python
## Each feature (column) is scaled in its own terms...
## All of the features should be normalized in order to have the compatible data
x_mean=X.mean()
x_std= X.std()
X=(X-x_mean)/x_std
```

[13]:

```python
## Now we have almost zero-mean data with unit variance, as shown below
X.describe()
```

[13]:

```
       Global_active_power  Global_reactive_power       Voltage  \
count         2.075259e+06           2.075259e+06  2.075259e+06
mean         -9.357604e-13           5.003686e-13 -5.873640e-11
std           1.000000e+00           1.000000e+00  1.000000e+00
min          -9.666490e-01          -1.104453e+00 -5.478824e+00
25%          -7.439309e-01          -6.759364e-01 -5.652359e-01
50%          -4.393591e-01          -1.938547e-01  3.731532e-02
75%           4.077311e-01           6.096149e-01  6.274429e-01
max           9.546788e+00           1.130469e+01  4.134043e+00

       Global_intensity  Sub_metering_1  Sub_metering_2
count      2.075259e+06    2.075259e+06    2.075259e+06
mean      -1.997329e-13   -2.570006e-14    2.134187e-13
std        1.000000e+00    1.000000e+00    1.000000e+00
min       -1.002552e+00   -1.834888e-01   -2.244450e-01
25%       -7.308426e-01   -1.834888e-01   -2.244450e-01
50%       -4.138488e-01   -1.834888e-01   -2.244450e-01
75%        4.012781e-01   -1.834888e-01   -5.159822e-02
```

```
max        9.911092e+00    1.420877e+01    1.360330e+01
```

[14]:
```
##Adding intercept row
X["intercept"]=1
```

[15]:
```
X
```

[15]:
```
                     Global_active_power  Global_reactive_power   Voltage  \
dt
2006-12-16 17:24:00             2.973748               2.627216 -1.863517
2006-12-16 17:25:00             4.062592               2.787910 -2.239335
2006-12-16 17:26:00             4.075917               3.341411 -2.344936
2006-12-16 17:27:00             4.089242               3.377121 -2.205169
2006-12-16 17:28:00             2.450266               3.609234 -1.602618
...                                  ...                    ...       ...
2010-11-26 20:58:00            -0.138594              -1.104453 -0.127299
2010-11-26 20:59:00            -0.140498              -1.104453 -0.260854
2010-11-26 21:00:00            -0.146209              -1.104453 -0.316761
2010-11-26 21:01:00            -0.150016              -1.104453 -0.354032
2010-11-26 21:02:00            -0.151919              -1.104453 -0.400621

                     Global_intensity  Sub_metering_1  Sub_metering_2  \
dt
2006-12-16 17:24:00          3.118368       -0.183489       -0.051598
2006-12-16 17:25:00          4.159919       -0.183489       -0.051598
2006-12-16 17:26:00          4.159919       -0.183489        0.121249
2006-12-16 17:27:00          4.159919       -0.183489       -0.051598
2006-12-16 17:28:00          2.529665       -0.183489       -0.051598
...                               ...             ...             ...
2010-11-26 20:58:00         -0.142140       -0.183489       -0.224445
2010-11-26 20:59:00         -0.142140       -0.183489       -0.224445
2010-11-26 21:00:00         -0.187425       -0.183489       -0.224445
2010-11-26 21:01:00         -0.187425       -0.183489       -0.224445
2010-11-26 21:02:00         -0.187425       -0.183489       -0.224445

                     intercept
dt
2006-12-16 17:24:00          1
2006-12-16 17:25:00          1
2006-12-16 17:26:00          1
2006-12-16 17:27:00          1
2006-12-16 17:28:00          1
...                        ...
2010-11-26 20:58:00          1
2010-11-26 20:59:00          1
2010-11-26 21:00:00          1
2010-11-26 21:01:00          1
```

```
2010-11-26 21:02:00         1

[2075259 rows x 7 columns]
```

[16]:
```python
## You can observe the shape of data by
print(X.shape)
print(y.shape)
```

```
(2075259, 7)
(2075259,)
```

[17]:
```python
# Get the sumber of samples
N=X.shape[0]

# Define the identity matrix
I=np.identity(X.shape[1])
indx_intercept=X.shape[1]-1
I[indx_intercept][indx_intercept]=0
I
```

[17]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0.]])
```

[18]:
```python
## Closed form solution and optimal linear regressor

# Define lambda here:
lam = 1/N # change the value

start1 = time.time()
## Calculate the closed-form solution here:
closed_form_sol= np.linalg.inv(X.T @ X + lam*N*I) @ X.T @ y
end1 = time.time()


reg = linear_model.Ridge(alpha=lam)

start2 = time.time()
## Find the optimal linear regressor here:
reg.fit(X,y)
end2 = time.time()

# Show the running time for the closed-form approach and the itterative algorithm
print(end1-start1)
```

```
print(end2-start2)
```

```
0.2887868881225586
0.04687929153442383
```

[19]:
```
## Show the optimal linear regressor based on the colsed-form solution
closed_form_sol
```

[19]:
```
0    42.585299
1     0.198422
2    -0.623129
3   -35.399833
4    -2.471475
5    -2.236698
6     6.458447
dtype: float64
```

[20]:
```
## Show the optimal linear regressor based on the itterative algorithm
reg.coef_
```

[20]:
```
array([ 42.60868266,   0.19889588,  -0.62345423, -35.42364539,
        -2.47128404,  -2.23650622,   0.        ])
```

[21]:
```
reg.intercept_
```

[21]:
```
6.458447357118318
```

[22]:
```
## Show the estimated y (w^T*X) based on the closed-form solution
a=np.dot(X, closed_form_sol)
a
```

[22]:
```
array([24.95811535, 34.72220112, 35.07867491, ...,  7.80064554,
        7.66174181,  7.60970853])
```

[23]:
```
## Show the estimated y (w^T*X) based on the itterative algorithm
reg.predict(X)
```

[23]:
```
array([24.95520363, 34.7201471 , 35.07726248, ...,  7.80119093,
        7.6622103 ,  7.61014766])
```

[24]:
```
## Check the gap between closed-form solution and the itterative one
reg.predict(X)- a
```

[24]:
```
array([-0.00291172, -0.00205403, -0.00141243, ...,  0.00054539,
        0.00046848,  0.00043913])
```

[ ]:
```

```

Part (c):Greenhouse gas observing network

[13]:
```
##imports from libraries
import pandas as pd
import numpy as np
import time
import glob
from sklearn import linear_model
```

[14]:
```
## Load data and preprocessing
## Preprocessing of data
# Load data here:

## reading dataset of ''Greenhouse gas observing network'' :

# get the absolute path of all Excel files
allExcelFiles = glob.glob("ghg_data/*.dat") #This is how to upload all dataset
data= pd.DataFrame()
# read all Excel files at once
for excelFile in allExcelFiles:
    pd_new=pd.read_csv(excelFile, sep=" ", header=None)
    data= pd.concat([data,pd_new],axis=1)
```

[15]:
```
## Transpose the data
data=data.T
```

[16]:
```
data.head()
```

[16]:
```
          0         1         2         3         4         5         6  \
0  0.174245  0.451203  0.224816  0.007046  0.006845  0.000118  0.289336
1  0.081913  0.027627  0.000447  0.000126  0.000121  0.000122  0.000314
2  0.053268  0.007294  0.030626  0.001596  0.001145  0.000122  0.016035
3  0.031948  0.000845  0.002176  0.000456  0.000210  0.000118  0.001872
4  0.016341  0.000120  0.000117  0.000120  0.000121  0.000121  0.000117

          7         8         9        10        11        12        13  \
0  0.013722  0.000198  0.000118  0.000118  0.000118  0.000118  0.000118
1  0.000120  0.000122  0.000122  0.000122  0.000122  0.000122  0.000122
2  0.001269  0.000122  0.000122  0.000122  0.000122  0.000122  0.000122
3  0.000206  0.000181  0.000119  0.000120  0.000123  0.000120  0.000120
4  0.000121  0.000121  0.000121  0.000121  0.000121  0.000121  0.000121

          14        15  16
0   1.399586  41.06452 NaN
1  14.964580  10.55327 NaN
2   2.585262  19.69646 NaN
3   1.631613  14.36786 NaN
4   2.230263  12.09993 NaN
```

```
[17]: ## Find the number of 'nan' in each column:
data.isnull().sum()
```

```
[17]: 0          0
1          0
2          0
3          0
4          0
5          0
6          0
7          0
8          0
9          0
10         0
11         0
12         0
13         0
14         0
15         0
16    954894
dtype: int64
```

```
[18]: ## Describe the data:
## count:  The number of not-empty values.
data.describe()
```

```
[18]:                   0             1             2             3             4  \
count  955167.000000  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05
mean        0.086227  7.560077e-01  4.286055e+00  6.466132e-01  8.688644e-02
std         0.286165  1.673985e+00  1.431162e+01  1.950537e+00  3.179971e-01
min         0.000100  1.000000e-18  1.694066e-09  1.000000e-18  1.000000e-18
25%         0.001790  3.915318e-02  4.286139e-02  5.715804e-03  1.263632e-03
50%         0.013318  2.106530e-01  4.325431e-01  7.316270e-02  1.331521e-02
75%         0.057204  7.134039e-01  2.283035e+00  4.128565e-01  5.773122e-02
max         9.983529  6.515823e+01  7.715252e+02  5.937678e+01  1.533314e+01

                  5             6             7             8              9  \
count  955167.000000  9.551670e+05  9.551670e+05  9.551670e+05  955167.000000
mean        0.021034  1.462339e+01  5.690690e+00  1.340781e+00       0.725947
std         0.098480  3.064796e+01  1.391971e+01  3.991153e+00       2.483712
min         0.000090  1.000000e-18  1.000000e-18  3.991246e-07       0.000080
25%         0.000122  4.153387e-01  2.586206e-02  3.184813e-03       0.000121
50%         0.000508  4.012232e+00  8.903016e-01  2.350419e-01       0.002117
75%         0.007060  1.544770e+01  5.302508e+00  1.007173e+00       0.185700
max         5.812891  1.136640e+03  4.581584e+02  1.598840e+02     119.611900

                 10             11             12             13  \
```

```
count  9.551670e+05  955167.000000  955167.000000  955167.000000
mean   2.012452e+00      21.149932       0.346534       3.856382
std    7.173313e+00      78.428447       1.796953      24.096129
min    2.646978e-11       0.000100       0.000100       0.000100
25%    1.383976e-04       0.000122       0.000120       0.000121
50%    8.943601e-02       0.076816       0.000155       0.002238
75%    1.334367e+00       6.032813       0.019189       0.353177
max    3.404201e+02    2250.325000      60.986410    1168.199000

                  14             15          16
count  955167.000000  955167.000000  273.000000
mean        2.002392      44.386022   61.157457
std         3.120457      63.730039   57.701962
min         0.000100       0.000030    0.261290
25%         0.163810      10.461565   21.975480
50%         1.130618      27.327980   40.747870
75%         2.483097      54.760060   80.891400
max        87.619210    1555.829000  287.092500
```

[21]:
```python
## Define the first 15 columns as X:
X=data.iloc[:,0:15]
```

[22]:
```python
X
```

[22]:
```
            0         1         2         3         4         5          6  \
0    0.174245  0.451203  0.224816  0.007046  0.006845  0.000118   0.289336
1    0.081913  0.027627  0.000447  0.000126  0.000121  0.000122   0.000314
2    0.053268  0.007294  0.030626  0.001596  0.001145  0.000122   0.016035
3    0.031948  0.000845  0.002176  0.000456  0.000210  0.000118   0.001872
4    0.016341  0.000120  0.000117  0.000120  0.000121  0.000121   0.000117
..        ...       ...       ...       ...       ...       ...        ...
322  0.007924  0.771132  0.584380  1.406428  0.041288  0.002193  70.692330
323  0.005464  0.379531  1.213877  0.899115  0.029752  0.001508  43.225760
324  0.005655  0.265967  0.369848  0.136567  0.019661  0.000348  46.614140
325  0.024174  2.004192  1.963009  0.180919  0.252553  0.000461  49.422240
326  0.037313  1.481214  2.124051  0.522502  0.145488  0.000677  47.113240

             7         8         9        10        11        12        13  \
0     0.013722  0.000198  0.000118  0.000118  0.000118  0.000118  0.000118
1     0.000120  0.000122  0.000122  0.000122  0.000122  0.000122  0.000122
2     0.001269  0.000122  0.000122  0.000122  0.000122  0.000122  0.000122
3     0.000206  0.000181  0.000119  0.000120  0.000123  0.000120  0.000120
4     0.000121  0.000121  0.000121  0.000121  0.000121  0.000121  0.000121
..         ...       ...       ...       ...       ...       ...       ...
322  81.987630  2.924276  0.001704  0.308882  0.283169  0.001614  0.032339
323  29.520890  2.677943  0.002806  1.285476  0.616680  0.002621  0.052846
324  50.773960  5.344912  0.001821  1.810855  0.710596  0.001506  0.053241
```

```
325  10.605850  6.674366  0.001557  0.559978  0.443013  0.001108  0.046486
326  45.950990  9.650680  0.001790  2.573767  0.482897  0.001347  0.048820

            14
0     1.399586
1    14.964580
2     2.585262
3     1.631613
4     2.230263
..         ...
322   1.283524
323   0.855096
324   0.834208
325   2.019726
326   2.748821

[955167 rows x 15 columns]
```

[23]:
```
## Define the column 16 as y:
y=data.iloc[:,15]
```

[24]:
```
y
```

[24]:
```
0       41.06452
1       10.55327
2       19.69646
3       14.36786
4       12.09993
          ...
322     97.56088
323     86.11260
324    107.13940
325     57.48664
326     96.99753
Name: 15, Length: 955167, dtype: float64
```

[25]:
```
X.describe()
```

[25]:
```
                   0             1             2             3             4  \
count  955167.000000  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05
mean        0.086227  7.560077e-01  4.286055e+00  6.466132e-01  8.688644e-02
std         0.286165  1.673985e+00  1.431162e+01  1.950537e+00  3.179971e-01
min         0.000100  1.000000e-18  1.694066e-09  1.000000e-18  1.000000e-18
25%         0.001790  3.915318e-02  4.286139e-02  5.715804e-03  1.263632e-03
50%         0.013318  2.106530e-01  4.325431e-01  7.316270e-02  1.331521e-02
75%         0.057204  7.134039e-01  2.283035e+00  4.128565e-01  5.773122e-02
max         9.983529  6.515823e+01  7.715252e+02  5.937678e+01  1.533314e+01
```

	5	6	7	8	9 \
count	955167.000000	9.551670e+05	9.551670e+05	9.551670e+05	955167.000000
mean	0.021034	1.462339e+01	5.690690e+00	1.340781e+00	0.725947
std	0.098480	3.064796e+01	1.391971e+01	3.991153e+00	2.483712
min	0.000090	1.000000e-18	1.000000e-18	3.991246e-07	0.000080
25%	0.000122	4.153387e-01	2.586206e-02	3.184813e-03	0.000121
50%	0.000508	4.012232e+00	8.903016e-01	2.350419e-01	0.002117
75%	0.007060	1.544770e+01	5.302508e+00	1.007173e+00	0.185700
max	5.812891	1.136640e+03	4.581584e+02	1.598840e+02	119.611900

	10	11	12	13 \
count	9.551670e+05	955167.000000	955167.000000	955167.000000
mean	2.012452e+00	21.149932	0.346534	3.856382
std	7.173313e+00	78.428447	1.796953	24.096129
min	2.646978e-11	0.000100	0.000100	0.000100
25%	1.383976e-04	0.000122	0.000120	0.000121
50%	8.943601e-02	0.076816	0.000155	0.002238
75%	1.334367e+00	6.032813	0.019189	0.353177
max	3.404201e+02	2250.325000	60.986410	1168.199000

	14
count	955167.000000
mean	2.002392
std	3.120457
min	0.000100
25%	0.163810
50%	1.130618
75%	2.483097
max	87.619210

[26]:
```
## Each feature (column) is scaled in its own terms...
## All of the features should be normalized in order to have the compatible data
x_mean=X.mean()
x_std= X.std()
X=(X-x_mean)/x_std
```

[28]:
```
## Now we have almost zero-mean data with unit variance, as shown below
X.describe()
```

[28]:

	0	1	2	3	4 \
count	9.551670e+05	9.551670e+05	9.551670e+05	9.551670e+05	9.551670e+05
mean	-6.280408e-16	1.954501e-14	5.468620e-15	3.172296e-15	-1.345253e-14
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	-3.009701e-01	-4.516217e-01	-2.994808e-01	-3.315052e-01	-2.732303e-01
25%	-2.950629e-01	-4.282325e-01	-2.964859e-01	-3.285748e-01	-2.692566e-01
50%	-2.547797e-01	-3.257824e-01	-2.692576e-01	-2.939962e-01	-2.313582e-01

```
75%   -1.014205e-01 -2.545055e-02 -1.399576e-01 -1.198422e-01 -9.168391e-02
max    3.458598e+01  3.847241e+01  5.360952e+01  3.010974e+01  4.794463e+01

                  5             6             7             8             9   \
count  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05
mean   3.493962e-15  1.553910e-14  4.492307e-14  1.388594e-14  1.290975e-14
std    1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min   -2.126652e-01 -4.771407e-01 -4.088225e-01 -3.359383e-01 -2.922509e-01
25%   -2.123431e-01 -4.635888e-01 -4.069645e-01 -3.351404e-01 -2.922343e-01
50%   -2.084247e-01 -3.462272e-01 -3.448627e-01 -2.770476e-01 -2.914310e-01
75%   -1.418943e-01  2.689623e-02 -2.788722e-02 -8.358686e-02 -2.175161e-01
max    5.881257e+01  3.660983e+01  3.250555e+01  3.972366e+01  4.786625e+01

                 10            11            12            13            14
count  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05  9.551670e+05
mean   5.549454e-16 -1.156706e-14 -4.785827e-15 -8.003317e-15  2.771586e-15
std    1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min   -2.805471e-01 -2.696704e-01 -1.927898e-01 -1.600374e-01 -6.416665e-01
25%   -2.805278e-01 -2.696701e-01 -1.927784e-01 -1.600366e-01 -5.892029e-01
50%   -2.680792e-01 -2.686922e-01 -1.927591e-01 -1.599487e-01 -2.793740e-01
75%   -9.452882e-02 -1.927505e-01 -1.821668e-01 -1.453846e-01  1.540493e-01
max    4.717592e+01  2.842304e+01  3.374595e+01  4.832073e+01  2.743727e+01
```

[29]:

```python
##Adding intercept row
X["intercept"]=1
```

[30]:

```python
X
```

[30]:

```
            0         1         2         3         4         5         6  \
0    0.307578 -0.182083 -0.283772 -0.327893 -0.251705 -0.212380 -0.467700
1   -0.015077 -0.435118 -0.299450 -0.331441 -0.272849 -0.212344 -0.477130
2   -0.115176 -0.447264 -0.297341 -0.330687 -0.269629 -0.212343 -0.476617
3   -0.189677 -0.451117 -0.299329 -0.331271 -0.272570 -0.212381 -0.477080
4   -0.244216 -0.451550 -0.299473 -0.331443 -0.272850 -0.212356 -0.477137
..        ...       ...       ...       ...       ...       ...       ...
322 -0.273629  0.009035 -0.258648  0.389541 -0.143391 -0.191315  1.829451
323 -0.282225 -0.224899 -0.214663  0.129453 -0.179668 -0.198266  0.933255
324 -0.281558 -0.292739 -0.273638 -0.261490 -0.211403 -0.210047  1.043813
325 -0.216844  0.745637 -0.162319 -0.238752  0.520969 -0.208907  1.135438
326 -0.170929  0.433222 -0.151066 -0.063629  0.184285 -0.206709  1.060098

            7         8         9        10        11        12        13  \
0   -0.407837 -0.335889 -0.292235 -0.280531 -0.269670 -0.192779 -0.160037
1   -0.408814 -0.335908 -0.292234 -0.280530 -0.269670 -0.192777 -0.160037
2   -0.408731 -0.335908 -0.292234 -0.280530 -0.269670 -0.192778 -0.160037
3   -0.408808 -0.335893 -0.292235 -0.280530 -0.269670 -0.192778 -0.160037
4   -0.408814 -0.335908 -0.292235 -0.280530 -0.269670 -0.192778 -0.160037
```

```
..        ...       ...       ...       ...       ...       ...       ...
322  5.481217  0.396751 -0.291597 -0.237487 -0.266061 -0.191947 -0.158699
323  1.711976  0.335031 -0.291153 -0.101345 -0.261809 -0.191387 -0.157848
324  3.238809  1.003252 -0.291550 -0.028104 -0.260611 -0.192008 -0.157832
325  0.353108  1.336352 -0.291656 -0.202483 -0.264023 -0.192229 -0.158112
326  2.892324  2.082080 -0.291563  0.078250 -0.263515 -0.192096 -0.158016

           14  intercept
0   -0.193179          1
1    4.153939          1
2    0.186790          1
3   -0.118822          1
4    0.073025          1
..        ...        ...
322 -0.230373          1
323 -0.367669          1
324 -0.374363          1
325  0.005555          1
326  0.239205          1

[955167 rows x 16 columns]
```

[31]:
```python
# Get the sumber of samples
N=X.shape[0]

# Define the identity matrix
I=np.identity(X.shape[1])
indx_intercept=X.shape[1]-1
I[indx_intercept][indx_intercept]=0
I
```

[31]:
```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

[32]:
```
y
```

[32]:
```
0       41.06452
1       10.55327
2       19.69646
3       14.36786
4       12.09993
          ...
322     97.56088
323     86.11260
324    107.13940
325     57.48664
326     96.99753
Name: 15, Length: 955167, dtype: float64
```

[34]:
```
## Closed form solution and optimal linear regressor

# Define lambda here:
lam = 1/N # change the value

start1 = time.time()
## Calculate the closed-form solution here:
closed_form_sol= np.linalg.inv(X.T @ X + lam*N*I) @ X.T @ y
end1 = time.time()


reg = linear_model.Ridge(alpha=lam)
start2 = time.time()
## Find the optimal linear regressor here:
reg.fit(X,y)
end2 = time.time()

# Show the running time for the closed-form approach and the itterative algorithm
print(end1-start1)
print(end2-start2)
```

```
0.2602508068084717
0.08037304878234863

/Users/zinatb/opt/anaconda3/lib/python3.9/site-
packages/sklearn/utils/validation.py:1688: FutureWarning: Feature names only
support names that are all strings. Got feature names with dtypes: ['int',
'str']. An error will be raised in 1.2.
  warnings.warn(
```

[35]:
```
## Show the optimal linear regressor based on the colsed-form solution
closed_form_sol
```

[35]:
```
0      0.315069
1      1.130347
2      9.497597
3      0.933297
4     -0.018555
5      0.197538
6     20.295841
7      8.957941
8      2.348403
9      0.991899
10     4.428147
11    54.059702
12     0.708045
13    16.270884
14     2.140556
15    44.386022
dtype: float64
```

[36]:
```
## Show the optimal linear regressor based on the itterative algorithm
reg.coef_
```

[36]:
```
array([ 3.15073759e-01,  1.13035094e+00,  9.49760831e+00,  9.33293843e-01,
       -1.85579046e-02,  1.97542222e-01,  2.02958645e+01,  8.95794891e+00,
        2.34840454e+00,  9.91891331e-01,  4.42814084e+00,  5.40597641e+01,
        7.08043606e-01,  1.62708995e+01,  2.14056105e+00,  0.00000000e+00])
```

[37]:
```
reg.intercept_
```

[37]:
```
44.38602157582871
```

[38]:
```
## Show the estimated y (w^T*X) based on the closed-form solution
a=np.dot(X, closed_form_sol)
a
```

[38]:
```
array([ 8.03977633, 16.60535554,  8.10000377, ..., 75.6324881 ,
       54.53543519, 79.21461835])
```

[40]:
```
## Show the estimated y (w^T*X) based on the itterative algorithm
reg.predict(X)
```

[40]:
```
array([ 8.03974402, 16.605343  ,  8.09997038, ..., 75.63251754,
       54.53544874, 79.21465131])
```

[41]:
```
## Check the gap between closed-form solution and the itterative one
reg.predict(X)- a
```

[41]:
```
array([-3.23132583e-05, -1.25394476e-05, -3.33955340e-05, ...,
        2.94360484e-05,  1.35503216e-05,  3.29615925e-05])
```

```
[ ]:
```

d) How would you address even bigger datasets?

Ans:

There are several approaches that can be applied to address the scalability problem in selecting the optimal linear regressor when dealing with larger datasets like

Stochastic gradient descent (SGD): SGD is an iterative optimization algorithm that can be used to find the optimal coefficients in a linear regression model. It works by randomly selecting one observation at a time and updating the coefficients based on the gradient of the loss function with respect to the coefficients. SGD is often used in large-scale machine learning problems and can be more computationally efficient than the closed-form expression.

Mini-batch gradient descent: Mini-batch gradient descent is a variation of SGD that uses a small subset of the data, called a mini-batch, to update the coefficients in each iteration. This can help to balance the trade-off between the computational efficiency of SGD and the accuracy of batch gradient descent, which uses the entire dataset in each iteration.

Dimensionality reduction: Dimensionality reduction is a pre-processing step that can be used to reduce the number of predictors in the linear regression model. This can be done by removing redundant or highly correlated predictors, or by transforming the predictors into a lower-dimensional space using techniques such as principal component analysis (PCA). Reducing the number of predictors can help to mitigate the scalability issue in finding the optimal linear regressor.